RESEARCH PROJECT
FINAL REPORT

Student: Adriana-Eliza Cozma
University of Stuttgart
Faculty of Computer Science, Electrical Engineering and Information Technology
Study programme: M.Sc. Computer Science

Coordinator: Prof. Dr. rer. nat. Marc Toussaint
## Contents

1 Motivation .................................................. 3

2 Subsystems .................................................. 3
   2.1 Racer .................................................. 3
      2.1.1 Short presentation .................................. 3
   2.2 OptiTrack .............................................. 4
      2.2.1 Short presentation .................................. 4
   2.3 Experimental Controller ............................... 5
      2.3.1 Short presentation .................................. 5
      2.3.2 How it was exploited ............................... 5
      2.3.3 Challenges .......................................... 7
   2.4 ZeroMQ ................................................ 7
      2.4.1 Short presentation .................................. 7
      2.4.2 How it was exploited ............................... 7
      2.4.3 Challenges .......................................... 9

3 The learning process ...................................... 9

4 Conclusion .................................................. 10
1 Motivation

When applying reinforcement learning algorithms in the context of simulation, the experiments one conducts are only limited by the processing power of the computer at hand and by the available time. When it comes to using the same approaches on the real system, the strategies derived from theory one would like to use are not an easy, perfect fit anymore.

In simulation, the execution flow is continuous and the entire learning process develops naturally, as a sequence of learning phases, without intermediate delays. Moreover, for a learning scenario where an episode ends when the agent falls, one does not need to worry about the agent actually falling and the consequences of this happening. On the other hand, for a learning experiment run in the real world, one has to find solutions that overcome these limitations. This research project aims to offer a considerable support with regard to this matter, by creating a setup where the learning process is continuous and the agent does not fall. This task is accomplished using an outer (also called experimental) controller, which constantly monitors the agent’s pose. It decides whether the agent is stable or not and sends it corresponding commands, such that it does not fall.

2 Subsystems

How does the whole system work? Let’s break it down so every component is better understood. The whole system is made of subsysyms, as presented in Figure [1]. Racer plays the part of the agent, OptiTrack streams data regarding Racer’s pose, the experimental controller monitors it and collects the learning data and ZeroMQ provides the communication between the agent and its outer controller.

In the following sections each subsystem is briefly described. This research project only focuses on the experimental controller and ZeroMQ library. The way they work and were used within this research project is also presented. At the end of their corresponding section, some of the encountered challenges are mentioned. These can be considered as open questions and matters that can be improved. Before drawing the conclusions, the entire learning process is detailed.

2.1 Racer

2.1.1 Short presentation

Racer is a wheeled inverse pendulum, which moves in $xz$ plane. More details about its components (battery, motors, encoders etc.) and how its dynamics is modeled can be found in $mlr\_students$ git repository $\rightarrow$ $racerRobust$ branch $\rightarrow$ $mlr/share/projects/racer/wiki$. From now on $mlr\_students$ git repository $\rightarrow$ $racerRobust$ branch $\rightarrow$ $mlr/share/projects/racer$ will be referred to as $path$.

Several threads are running in parallel on Racer. For example, in $path/racerRobust/main.cpp$, which is the code that runs on Racer in the context of this research project (referred to in the following as Racer’s code), $RoboClawModule$ deals with the motors, encoders and battery. $IMU\_poller$ reads the IMU data, i.e. gyroscope and accelerometer data. $KalmanFilter$ thread computes Racer’s
OptiTrack streams data to a publisher node, which publishes it to the \textit{tf} topic. The experimental controller subscribes to this topic in order to have access to this data. The experimental controller and Racer communicate via ZeroMQ messages, in a bidirectional way. A more detailed presentation regarding this data flow will be delivered in the following sections.

estimated state based on this data. With the help of \textit{GamepadInterface}, Racer can be controlled via the gamepad. The last used thread is \textit{ZeroMQ}, which deals with the communication between Racer and the experimental controller.

These threads share variables, having read and write access to them. The access has to be explicitly stated when the threads are declared, using instances of \textit{ACCESS} class.

More information regarding the use of threads is available in the comments within \textit{mlr_students} git repository $\rightarrow$ \textit{racerRobust} branch $\rightarrow$ mlr/share/src/Core/thread.h and \textit{thread.cpp} and in the last part of the guide - \textit{path/wiki/mlr.pdf}.

2.2 OptiTrack

2.2.1 Short presentation

OptiTrack is a motion capture system, providing high-speed tracking via its cameras. A clearer and better documented presentation of OptiTrack is available at \textit{mlr_students} repository $\rightarrow$ \textit{optitrack} branch \textit{mlr/usr/VI}.

In short, OptiTrack consists of cameras, markers and a motion capture software, called \textit{Motive}. After the cameras are arranged such that the volume they capture fits the task at hand and the markers are placed as desired, the system needs to be calibrated.
Tracking data streamed by *Motive* is handled by *mocap_optitrack* ROS node. It publishes the data in *tf* topic, where it can be easily accessed by other nodes within ROS environment. More information about how to set up and properly use OptiTrack and *mocap_optitrack* can be found following the above mentioned path.

### 2.3 Experimental Controller

#### 2.3.1 Short presentation

In our application, the experimental controller plays the part of an outer supervisor. Having access to OptiTrack data, it can monitor Racer’s pose, checking if it is stable or not, based on some criteria. Using ZeroMQ functions, it communicates with Racer. It handles the receiving and storing of learning data, by saving it into a file. If the reinforcement learning experiment requires noise to be added to the policy, the experimental controller also takes care of this matter. It basically controls the conducted experiment, thus its name was derived.

The experimental controller runs on a ROS node (for simplicity and hopefully, without loss of understanding, we will refer to this node also as experimental controller). In Section 2.2.1 we have briefly described how the flow of OptiTrack data is managed within ROS environment. As a short reminder, *mocap_optitrack* node publishes the incoming data in *tf* topic. The experimental controller is a subscriber to this topic, thus having access to OptiTrack data. Messages between publisher and subscriber node are passed to a callback function, in our code called *optitrackCallback*. The calls to this function are stopped when *Ctrl − C* is pressed. The frequency of calling the callback function is given by the publishing frequency, which is 120Hz.

#### 2.3.2 How it was exploited

OptiTrack streams data regarding each visible marker’s position and each created body’s pose (translation and orientation on *x*, *y*, *z* axes). In our setup, there is only one body created in *Motive* software, the one corresponding to Racer. The experimental controller needs to select only the data concerning Racer’s pose from all available data within *tf* topic.

The rotation information is converted from quaternion to Euler angles, i.e. roll, pitch, yaw, which are given in radians. In all conducted experiments, Racer’s body was created facing towards positive *x* axis, so it starts moving along *x*. Because of this choice, the experimental controller in our application is interested in Racer’s orientation on *y* axis, so in *pitch* angle. The angular velocity, i.e. *pitch* derivative, is also computed, being necessary in determining whether Racer is stable or not.

There are three types of messages the experimental controller could receive from Racer. It receives *Start!* and *Stop!* messages, when Racer starts and stops running, respectively. Racer also sends learning data messages when it is in the exploration phase. This type of message, in our reinforcement learning scenario, contains Racer’s state, the corresponding action, encoder values and the time when this information was retrieved. The experimental controller saves the learning data messages into a file along with *pitch* value, translation information (provided by OptiTrack),
time and current step, only when Racer is stable. Being unstable is equivalent to the end of an episode.

If Racer is running, its stability is decided based on its pitch angle and angular velocity. The stateStable variable stores the stability information. If the corresponding absolute values are beyond certain thresholds, the experimental controller concludes that Racer is unstable and sends it a Not stable message. The experimental controller confirms that Racer is stable only after it is seen in the stability region for some time. Accordingly, a message containing Stable and the current policy is sent to Racer. The stability region is also defined by the above mentioned thresholds, as illustrated in Figure 2, but it is not the opposite of the instability zone. The thresholds separating the two regions were initially picked by trial and error and then by using a more systematic way.

Because the working frequency of the experimental controller is higher than the frequency of received messages, Racer could change from stable to unstable faster than it can send learning data messages. This leads to episodes with no steps taken. Because we do not want to count it as a valid episode, the written variable is used to show that at least one learning data message was saved in the file per episode.

The experimental controller sends just one message to Racer regarding its stability, when stateStable changes its value. This modification is enclosed into the first flag.

After the desired number of episodes is reached, two other programs are run within the experimental controller’s main() function, making use of the system() feature. The first program, DataEval, processes the file containing the learning data. The second run program is policySearch, which updates the policy based on the current learning data. A more detailed description of these programs will be offered in Section 3. After these programs run, the experimental controller starts overseeing Racer’s stability and collecting learning data again until the desired number of updates
is reached (or any other user-dependent condition is fulfilled). We use this approach to obtain a continuous execution flow. This ensures that no delays, which may derive from going from one program to another, are introduced.

2.3.3 Challenges

Not a challenge for us, but for those who may work in this framework, the code might be a bit hard to digest. It uses several flags and managing various events, e.g. reading OptiTrack data, receiving and sending ZeroMQ messages, supervising Racer’s stability etc. At first glance, the code may look complex and a bit too difficult to understand. We guarantee though that it is in essence well structured. The wish for a continuous execution flow caused this apparent complexity of the code.

One challenge came from integrating ZeroMQ methods in ROS environment, but this problem will be discussed in Section 2.4.2.

2.4 ZeroMQ

2.4.1 Short presentation

In broad lines, ZeroMQ is a messaging library, which allows the transfer of messages through different types of sockets. Each kind of socket involves a specific communication scheme and requires a certain messaging pattern. The basic socket patterns one could choose from are: request-reply, publish-subscribe, pair, push-pull.

In order to get ZeroMQ running, you first need to install it and the easiest way to do that is by following the installation instructions from the official website, [1] and [2].

There are also some examples provided at [3] and [4], which show how different sockets work, considering a simple, straight-forward communication setup. Those interested in getting a better sense of ZeroMQ’s functionalities should definitely go through these examples. For an even more thorough and complete understanding, one should read the dedicated guide, available online [5].

2.4.2 How it was exploited

ZeroMQ is the liaison between two main subsystems. It ensures the communication between Racer and the experimental controller. This connection is fulfilled by explicitly sending and receiving messages on both sides, through sockets. Racer handles the flux of messages with the help of ZeroMQ thread. The experimental controller calls corresponding receive and send functions within its code.

After some research into the characteristics of each socket type, we went with the pair pattern, which seems to be the most appropriate choice for our implementation. The pair sockets design is a one-to-one connection, between only two peers, a client and a server, respectively [6]. The communication is bidirectional, asynchronous and any number of messages can be sent between
the peers. The socket type is passed as an argument when declaring it, as can be seen in Code Snippet[1] Each of the two peers has a socket. The server (experimental controller) listens on a certain port and the client (Racer) connects to it (the IP of the server is also required), as presented in Code Snippet[2]

Code Snippet 1: ZMQ_PAIR is the flag needed when declaring a pair type socket.

```c
static zmq::context_t context (1);
static zmq::socket_t socket (context, ZMQ_PAIR);
```

Code Snippet 2: (a) The server (here, experimental controller) binds to a port. 
(b) The client (here, Racer) connects to the same port. The server’s IP is also needed.

```
a) socket.bind("tcp://*:5090");
b) socket.connect("tcp://129.69.216.10:5090")
```

While testing ZeroMQ in a simple way (the experimental controller sends a message consisting of an integer value to Racer; Racer receives, increments and sends it back), we noticed that the received messages are empty on both sides. After a quite long and cumbersome investigation, the conclusion we reached had more the value of an assumption than of a clear, definite answer. Because the experimental controller runs on a ROS node, we speculated that there might be a conflict between ROS and ZeroMQ. The solution for this problem of receiving empty messages is to use a certain send function, namely `zmq_send` and not the counterpart of `socket.recv`, which is `socket.send`. Therefore, because each function requires a specific data type for the transmitted message, in our application the sent messages are of type `zmq_msg_t`, while the received messages are declared as `zmq::message_t`, as shown in Code Snippet[3]

Code Snippet 3: Different data types for sent and received messages, respectively.

```
zmq_msg_t mesS;
zmq::message_t mesR;
```

Another ZeroMQ feature we made use of is the ZMQ_NOBLOCK flag. It indicates that the operation, send or receive, should pe performed without blocking. The ZeroMQ thread running on Racer should not block if a message is not ready to be sent (or received), because it may have a received (or sent) message available on the socket, which needs proper handling. The same reasoning is valid also for the experimental controller. It should not block waiting for a message of any kind to be ready, because it also has other things to take care of (e.g. take data from OptiTrack and monitor Racer’s orientation). The direct effect of ZMQ_NOBLOCK flag is a continuous execution flow.

A ZeroMQ message is transmitted as a string. The maximum number of bytes one could broadcast at once is 64. If the message that needs to be sent exceeds this limit, one could consider splitting it and sending the resulting parts separately. We apply this solution in our implementation, for handling the large messages Racer sends to experimental controller.

In all conducted experiments, ZeroMQ performed well, providing a reliable and robust communication between Racer and the experimental controller. Testing revealed that all sent messages were received, with no data lost.
2.4.3 Challenges

The most challenging task when dealing with ZeroMQ was integrating it within the ROS environment, where the experimental controller runs. Finding a solution to the problem of receiving empty messages was quite difficult, mainly because we could not come across any example of ROS and ZeroMQ combined usage. Lacking any technical support regarding this issue, we were forced to try almost everything until overcoming this challenge.

3 The learning process

In this section details specific to our implementation, exclusively concerning the learning process will be mentioned. Some pieces of information related to this topic were already presented in the previous sections, but will be briefly resumed here for sake of completeness.

The aim is to learn a stabilization policy, by applying a policy gradient method. In our case, the gradient is approximated using linear regression. The considered policy is a parameterized stochastic Gaussian policy, linear with respect to its parameters, noted as $\theta$. When Racer is stable, it receives the current parameters from the experimental controller. In our implementation, the features vector is equivalent to Racer’s state vector, consisting of position along $x$ axis, angle from $z$ axis, derivative of position and derivative of angle. The action is sampled using $\text{sampleAction()}$ function, from $\text{LinearPolicy}$ class and no noise is added to it. The learning data Racer sends to the experimental controller includes its estimated state (provided by $\text{KalmanFilter}$ thread), the action, encoder data and the current time.

The experimental controller saves Racer’s learning data in a file, along with current $\text{pitch}$ value, translation information (both provided by OptiTrack), time and step. An episode ends when Racer becomes unstable. Its stability status is judged based on $\text{pitch}$ angle and angular velocity (derivative of $\text{pitch}$). The corresponding values are derived from the data OptiTrack provides, which is more reliable and robust compared to Racer’s sensors data. The stability and instability regions are outlined with respect to two thresholds. Irrespective of the sampled action, the chosen thresholds should prevent Racer from falling.

In the context of our reinforcement learning scenario, each episode corresponds to a noisy $\theta$, obtained from adding a Gaussian noise to an initial, base, $\theta$. The standard deviation of this noise in all our experiments is 0.1. The option of adding noise to base $\theta$ is selected by passing $\text{noise}$ as the second argument in the command line, when running $\text{ExpController}$. The policy parameters for the current episode are also saved in the file, before the learning data corresponding to the first step taken is written in.

After the desired number of episodes is achieved, the file containing learning data is processed by $\text{DataEval}$ program, which is called from $\text{ExpController}$ code, using $\text{system()}$ function. The learning data acquired for one step may contain more information than it is actually required by the chosen reinforcement learning algorithm. We wanted to separate the data acquisition phase from its preparation for the considered learning algorithm. This segregation ensures the flexibility of code. Consequently, $\text{DataEval}$ goes through the file the experimental controller creates and saves only the data of interest. It also collects all $\theta$ parameters into a separate file. Regarding the
learning data, for each episode, at each step, the encoder data at first step is subtracted from the current encoder information. Further processing on this data can be done, according to the learning task at hand.

After DataEval, policySearch program is run, also by using system() function. It goes through the learning data file created by DataEval and computes at each step the corresponding reward. In our implementation, the reward is determined taking into account the pitch value, encoder data, action and the fact that Racer did not fall. This assessment of the reward encourages Racer to be as straight as possible, try not to move at all, perform the least amount of effort and run for longer time (i.e. many steps), respectively. For each episode the expected return is computed and the corresponding noisy \( \theta \) is read from the file DataEval created. After going through all episodes, the gradient is evaluated by performing linear regression. The base \( \theta \) is then updated, using Rprop method. The previous gradient and learning rates are also read from a file, managed by policySearch. After the update, \( \theta \) is written back in the same file from where ExpController reads it. Now learning data can be collected again, this time corresponding to the updated parameters.

After around 30 updates, the parameters \( \theta \) stagnated, without any significant difference in their values with respect to the previous 10 update stages. The learning rates were also small, so no notable difference could have been expected for the next updates. It was definitely not a case of convergence, but still we tested the obtained policy and Racer fell right from the beginning. We assumed that a local optimum was reached instead and we decided to boost the learning rates. The parameters showed a considerable change in their values in the following 10 updates, but then they stagnated for 10 more. The policy was tested again, but with no successful results. Moreover, after a closer look in the learning data file, we noticed that the actions were small, between 0 and 1 before the boost in learning rates and reached values around 4 after the boost. These actions are definitely not enough to turn the motors, considering the fact that the commands given by PD law are usually in the hundreds. As a workaround, we multiplied the sampled action with a factor of 50 and then applied it, in the context of a new learning experiment. After 30 updates we observed the same trend in how \( \theta \) and the actions change. For confirmation, the obtained policy was tested, but with no promising results.

4 Conclusion

During this research project we managed to build a framework in which an experimental controller supervises the entire learning process. It constantly monitors Racer’s pose, based on reliable data, provided by OptiTrack motion capture system. It checks whether Racer is in the stability region or not and sends it corresponding commands. If Racer is stable, it starts exploring, by sampling actions from the current policy. If it is unstable, it needs to stabilize, with the help of a PD control law. The communication between Racer and the experimental controller is ensured by ZeroMQ messaging library. The experimental controller’s main duty is basically to prevent Racer from falling, which leads to a continuous learning process. It also handles the learning data acquisition and processing and the policy updates, maintaining a constant execution flow.
The proposed framework successfully manages to prevent Racer from falling, but with significant drawbacks. We believe that the exploration phase is limited by this framework. The learning is not done efficiently, because Racer is not allowed by the experimental controller to fully experience the results of the performed actions. Therefore, how much it learns is constrained by the chosen stability margins. On one hand, small thresholds guarantee that Racer will not fall, but suppress its learning ability. On the other hand, larger values for the stability margins are not enough to keep Racer from falling, although they may be more permissive in terms of how much Racer learns. Instead of accepting this compromise, we suggest another solution that could be implemented at hardware level. Our recommendation refers to a hardware addition that prevents Racer from falling, while still allowing it to explore freely. Thus, Racer could reach orientations that are forbidden, i.e. considered unstable, within our framework. Figure 3 illustrates a possible prototype for the proposed hardware solution.

Figure 3: A possible implementation of the proposed hardware solution, which could prevent Racer from falling and also let it explore other states.

We applied a policy gradient method to compute the stabilization policy. The gradient was estimated using linear regression. Another approach could be also tried out that would take better advantage of the collected learning data.

References


